Proceedings of the IASTED International Conference
AI and Soft Computing
August 9-11, Honolulu, Hawaii, USA

Verifying an AI Planner for an Autonomous Spacecraft

Ben Smith and Martin Feather

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
firstname.lastname@jpl.nasa.gov

Abstract—The six-day Remote Agent Experiment (RAX) on the Deep Space 1 mission will be the first time that an artificially intelligent agent (the *Remote Agent*) will control a NASA spacecraft. Successful completion of this experiment will open the way for AI-based autonomy technology on future missions. An important validation objective for RAX is implementation of a credible validation and verification strategy that will "scale up" to missions that make full use of spacecraft autonomy. One of the key components of the Remote Agent is an onboard planner and scheduler. Verifying autonomous planners raises several challenges. This paper describes those challenges and the verification methods we found effective.

1. INTRODUCTION

The Deep Space One (DS1) spacecraft, which launched October 24, 1998, is a technology validation mission. Unlike previous missions its primary objective is not to gather science observations, but to flight validate several new technologies. Successful validation of these technologies will remove a major obstacle to their use on more risk-averse science missions.

One of the new technologies is the Remote Agent (RA), an artificial-intelligence based architecture capable of autonomously commanding the spacecraft. The Remote Agent Experiment (RAX) will demonstrate autonomous operations of the DS1 spacecraft by the RA for a period of six days in the Spring of 1999. Successful flight validation of the RA will open the door to the use of autonomous spacecraft commanding technology on future science missions. A excellent description of the experiment design and validation objectives can be found in [1].

The Remote Agent (RA) consists of three components: an onboard planner, a smart-executive, and a mode identification and recovery engine (MIR). The onboard planner expands high-level goals, provided by the ground operators and onboard agents, into a plan that achieves the goals while obeying various safety, resource, and operational constraints. The smart executive carries out the plans while MIR looks for faults. If a fault occurs, the executive tries to recover within the constraints of the plan; otherwise it puts the spacecraft into a known safe state (possibly degraded) and requests a new plan that achieves the remaining goals from that state.

One of the Remote Agent's validation objectives is demonstrating a credible verification and validation (V&V) approach that will "scale up" from the experiment scope to missions that make full use of autonomy. Autonomous systems raise several verification challenges not faced by traditional flight software. This paper focuses on the challenges raised by the onboard planner and on the verification methods that we found effective.

The onboard planner must be able to generate a valid plan given a set of goals and an initial state. There are far too many combinations to test exhaustively. The challenge is to find a set of input test cases that is small enough to test and analyze tractably, yet still provides a high degree of confidence in the planner. This challenge was compounded for RAX by limited testing resources, both in terms of workforce and testbed availability.

We have taken a three-pronged approach to planner verification. First, we used a parameter-based approach to identify a test suite with reasonable coverage of the input space. The inputs are mapped onto independent parameters, and then the planning constraints are used to identify non-interacting or low-interaction regions of that space. This greatly reduces the number of parameter combinations that must be tested. Orthogonal arrays were used to design minimal-sized test suites with comprehensive coverage of the necessary parameter combinations.

Second, we developed a automated plan verification tool to increase the number of test cases that we can feasibly analyze.

Finally, we exploited the availability of lower-fidelity test beds to reduce the number of tests that must be run on the oversubscribed high-fidelity testbed. The planner is composed of mission- and platform-independent reasoning engines, and mission-dependent models. Almost all of the reasoning requirements can be verified on lower fidelity platforms. The high-fidelity platforms test the remaining performance and operating environment requirements.

The remainder of this paper is organized as follows. Sections 2 describes the Planner and Remote Agent Architecture in more detail. Section 3 describes the test-case selection methods. Section 4 discusses the automated verification tools and specification formalisms that we found effective. Section 5 describes

the testbed allocation approach and lessons for larger-scale testing efforts.

2. Remote Agent Architecture

The Remote Agent [2] consists of three components: Executive (EXEC), Planner/Scheduler (PS), and Mode Identification and Reconfiguration (MIR). The planner is given a set of high-level goals from the ground operations team and from the onboard navigation system. PS generates a plan that achieves the goals while obeying safety constraints and resource constraints. The plan is carried out by the smart

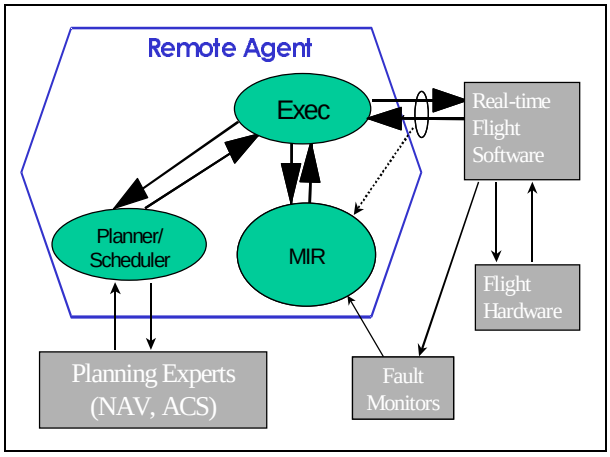

Figure 1: Remote Agent Architecture

executive. Faults are detected by the MIR engine. If the smart executive cannot resolve the fault within the constraints of the plan, it puts the spacecraft into a known (and possibly degraded) safe state and requests a new plan that achieves the remaining goals from that state.

The fundamental execution units in the plan are tokens and timelines. Each activity in a plan is defined by a token, though not every token is an executable activity. Tokens also track spacecraft states and resources. A timeline is a sequence of tokens that specifies the evolution of that state variable over time. The plan has several parallel timelines. The plan specifies start and end time windows for each token, and temporal constraints among the tokens (before, after, contains, etc).

The Remote Agent architecture is shown in Figure 2.

3. Test Case Selection and Coverage

The effectiveness of scenario-based testing depends largely on how well the scenarios cover the requirements. Good coverage requires not only that the test suite exercise each requirement, but that the test-suite provide high confidence that if the requirement is satisfied on the test suite that the requirement will also be satisfied on all of the untested inputs.

In addition to providing good coverage, the test-suite must have a manageable number of tests. Finding the right balance between coverage and test-suite size can be difficult, and may involve trading risk (coverage) for manageability. The manageability of a test-suite depends on the availability of appropriate test-beds, the running time of the suite, and the analysis effort it entails.

The module test-suites were designed using a parameter-based approach. The universe of possible input scenarios is characterized by a multi-dimensional parameter space. A given assignment of values to parameters specifies a unique scenario. The test suite consists of a subset of the possible parameter values. Three methods were used to achieve good coverage and manageability: abstracting the parameter space to focus on the relevant parameters and values, analyzing the planner models to identify independent regions of the parameter space and thereby reduce the number of parameter combinations that must be tested, and using orthogonal arrays to generate minimal-size test suites that cover those combinations.

Planner Test Case Selection

The planner takes as input a requested plan start time, an initial spacecraft state, a set of goals (some of which come from the onboard navigator), and a set of constraints. It generates a plan that begins at the requested start time and achieves the goals from the initial state while obeying the constraints. The constraints are specified in the plan model and are largely fixed. However, a few of them can be modified by changing parameter settings, and fewer still are defined as external functions provided by onboard systems. (Specifically, the duration and legality of spacecraft *slews* (turns) are defined as functions provided by the attitude control subsystem.) The constraint parameters and the behavior of the ACS function must both be treated as inputs.

The planner's behavior is strictly a function of its inputs. Its behavior does not depend on the order or timing of events that occur while it is planning[1]. This makes it a good candidate for parameter-based testing (e.g., [4]). The input space is characterized by a multi-dimensional parameter space. Each assignment of values to parameters identifies a single point in the input space. The planner is tested on a carefully chosen subset of parameter values, and the resulting plan is checked against a list of plan correctness requirements as discussed in Section 4.

The test-suite must have good coverage, as defined by some metric, but not be too large to run and analyze. Based on our experience a suite of 200 to 300 plan-request scenarios is about the upper limit for a one-person testing effort, assuming an automated scenario-runner and adequate plan analysis tools.

[1] The onboard goals and ACS constraint functions are invoked during planning, but they always give the same results for the same inputs regardless of when they are called or in what order.

A combination of approaches has proven effective for generating test-suites for the RAX planner. First, the parameter space is reduced by identifying equivalence classes of parameters and parameter values. The planner behavior is not expected to change qualitatively on inputs drawn from the same equivalence class, but is expected to change for inputs in different classes. Next, regions of high and low interaction in the reduced parameter space are identified by analyzing the planner model. Parameters from strongly interacting regions should be tested in combination, while fewer combinations must be tested from weakly interacting regions. Parameters from non-interacting regions can be tested independently. Finally, an orthogonal array-based algorithm generates a small (nearly minimal) size test-suite with comprehensive coverage of the identified parameter combinations.

Parameter Space Construction—The input space is characterized by a multi-dimensional parameter space such that there is a one-to-one correspondence between parameter settings and inputs. We term this the "true" parameter space. This space is infinite, and clearly infeasible for testing.

To produce a manageable number of test cases, it is first necessary to control the size of the parameter space. This was done by selecting parameters and parameter values that focus on aspects of the input space to which the planner is expected to be sensitive. We term this the "abstract" parameter space. Each parameter setting in this space specifies an equivalence class of inputs rather than a single input. The planner is expected to behave similarly on every input in a given class, but to have qualitatively different behavior for inputs drawn from different classes. Abstraction entails some risk, since there is no guarantee that the parameter space actually has these properties, but this risk is needed to construct a manageable test-suite.

Real or integer-valued parameters from the true space, and those with large numbers of values, were abstracted by selecting a small handful of discrete values to test. Where it was known which values were at boundaries of qualitative behavior regions of the planner, those boundary values were selected. In the other cases values were selected from the parameter's domain according to their expected distribution in operations. No attempt was made to select a statistically significant number of values. The abstraction makes several educated guesses, such as the qualitative behavior boundaries, the distribution of values in operations, and the number of parameter values to select.

The primary abstractions are as follows. The initial states are restricted to twelve canonical states that cover all of the qualitatively different initial states [all combinations of MICAS state (2), MICAS health (2), and final attitude (3)]. The planner is either insensitive to other variations, or those variations are not initial states that the exec would ever generate. The plan start time is restricted to ten boundary points: before, during, and after plan horizon boundaries; and before, during, and after op-nav windows.

Several parameters are set to fixed values. Most of these are design-time parameters that might change during testing and integration, but will not change during the experiment itself. The suite will be re-run if those parameters change. The remaining parameters control values that the planner does not reason about, but simply "passes through" to the executive. Testing a single arbitrary value is sufficient.

Scaling Up—There are a number of aspects to the planner and RA design that reduce the size of the parameter space, and thereby facilitate testing. It will be important to pay attention to these design decisions for future missions.

There are a vast number of possible initial states, but only ten or so that will occur in practice. By design, the RA reduces variability in the initial state for planning. The Executive places the spacecraft in a predefined state after any error that requires replanning. In nominal operations, the initial state for a given plan is the end-state of the previous plan. This allows execution to continue seamlessly from plan to plan. The planner model is designed so that every plan ends in a relatively quiescent state similar to the standby state. There are only a small number of qualitatively different end-states.

A full-scale science mission will have a few more standby states and end-states, but not many more. The number will be proportional to the product of the health-states tracked by the planner. If the planner covered the entire DS1 mission, it would track only three more health timelines: IPS health, MICAS high-voltage switch health, and RCS thruster health for a total of 32 initial states.

Test Suite Construction—The test suite must provide adequate coverage, according to some metric, yet have a manageable number of cases. We use a combination of two approaches. First, we use *orthogonal arrays* [4] to generate a minimal-sized test-suite in which every parameter value and every pair of values appears in at least one test case, and every parameter value appears in about the same number of cases.

This approach detects every bug caused by a single parameter value or by an interaction of two parameter values. It will detect only some bugs caused by interactions of three or more parameter values. The risk of this approach is that it assumes that the majority of bugs are due to one or two parameter values.

The PS test-suite was constructed using an orthogonal arrays approach. The RAX test-suite contains three sub-suites generated with orthogonal arrays: one for the twelve-hour experiment, and one each for the six day replan cases and the six-day back-to-back plans. The twelve-hour suite has 24 test cases (many of the above parameters are fixed for the twelve-hour

Table 1. DS1 Test-beds

Platform	*Fidelity*	*CPU*	*Hardware*	*Availability*	*CPU speed*
Spacecraft	Highest	Flight	Flight	1 for DS1	1:1
DS1 Testbed	High	Flight	Flight spares + DS1 simulators	1 for DS1	1:1
Hotbench	High	Flight	Flight spares + DS1 simulators	1 for DS1	1:1
Papabed	Med	Rad6k	DS1 simulators only	1 for DS1	4:1
Radbed	Low	Rad6k	RAX simulators only	1 for RAX	4:1
PowerPC	Lowest	PPC	RAX simulators only	2 for RAX	10:1
Unix	Minimal	Sparc Ultra	RAX simulators only	unlimited	40:1

- The flight CPU is a radiation hardened RS-6000 chip (Rad6k) running on the flight bus, memory, etc.
- The Papabed and Radbed run on a Rad6k chip bus, but have some non-flight bus and memory components.
- The PowerPC (PPC) is a non-hardened, off-the-shelf RS-6000 chip with higher clock speed than the Rad6k.
- The RAX simulators were written by the RAX team, and are of lower fidelity than the DS1 simulators.

experiment), and the other suites have about fifty cases each.

Coverage Metrics—Constructing this test suite required several assumptions and abstractions, and each of these introduces some uncertainty in the coverage. In order to assess this risk, we developed several orthogonal coverage metrics with which to evaluate the test suite and its assumptions.

One metric is whether there is at least one test case from each set of inputs for which the planner behavior is expected to be qualitatively different. Whether the planner behaves qualitatively similarly or differently to a set of inputs depends on the constraints in the planner model. If the inputs differ on plan elements that are interact strongly (have many constraints among them), then the planner behavior is likely to differ dramatically. If the inputs differ on weakly interacting plan elements, then the output plans are likely to be similar. With this metric, one identifies all the strongly interacting plan elements and the combinations of "true" parameter values that control these elements. The abstract parameter space should not have settings that correspond to these combinations. If it does, then settings in the abstract space do not correspond to equivalence classes where the behavior is qualitatively similar.

The test-suite coverage is measured with respect to this metric by identifying the combinations of abstract parameter values that control the strongly interacting regions. The test-suite should have at least one test case for each combination. The orthogonal array algorithm can be extended to include these test cases (e.g., [5]), or they can simply be appended to the test-suite with no attempt to minimize.

We performed a very rough interaction analysis to identify the most heavily interacting goals, initial states, and constraint parameters. The test suite contains at least a few test cases from each of these interaction regions. Additional work is needed to implement the interaction analysis metric.

A second metric is how well the test suite exercises all of the requirements. If a requirement is trivially satisfied for some test-case, then that case does not exercise the requirement. A third related metric is how well the test-suite exercises all of the knowledge in the plan model. The plan model consists of constraints of the form "if token A appears in the plan, then token B must also appear in the plan and be in the following temporal relation to A." Token A is called the master token, and B is called the target token. The constraint is exercised if and only if the master token appears in the plan. It is therefore easy to determine which constraints were exercised by examining the tokens in the resulting plan. As an additional check, the plan maintains temporal relations, which makes it possible to tell whether a master/target pair occurred by accident or as a result of exercising a constraint in the model. The coverage of the test-suite is proportional to the percentage of the total constraints exercised.

It is difficult to predict which inputs will exercise a given constraint or requirement, though one can often make a good guess. For the RAX planner, we use these two coverage metrics to measure the coverage of the suite after running it, and then add test-cases if needed. The third metric is analogous to code-covering metrics which are also used for post-hoc coverage analysis.

Single Variation Test Suites—The orthogonal array-generated test suite provides excellent coverage with a handful of test cases. However, it is difficult to identify which parameter caused a problem and file a meaningful bug report. To address this problem, we constructed a second suite of test-cases in which each case is identical to a baseline scenario in every parameter value but one. Since the planner is known to perform correctly on the baseline case, any problems are very likely to be caused by the one parameter that changed.

In practice, these "single variation" cases catch most of the initial bugs. The orthogonal-array suites are useful for identifying additional bugs once the single variation cases all pass.

4. ALLOCATING TESTS TO TEST BEDS

The DS1 flight project has a number of test beds ranging in fidelity (with respect to the spacecraft) and scarcity, as shown in Table 1. The highest fidelity platforms, such as the spacecraft itself, are scarce and testing time on them is limited. CPU speed decreases with increasing fidelity,[2] further limiting the effective testing time. Lower fidelity platforms are more numerous and have faster CPUs, but tests performed on them must be combined with a strong argument that additional fidelity will not change the outcome.

Exploiting speed and availability is crucial for the large planner test suites. In order to obtain adequate coverage, the test suite must have about 200 plan generation cases. Plans take four hours to generate on flight CPUs, but less than five minutes to generate on a Sparc Ultra/1 which means it would take over a month to run through the entire suite on a high-fidelity test-bed, but only sixteen hours on Unix.

High fidelity test beds are also the most difficult to configure and instrument for a given test, whereas lower fidelity test-beds are generally the easiest. This means that some tests can only be performed on lower fidelity test beds.

The planner is well suited to exploit low-fidelity testbeds. The planner consists of a domain-independent reasoning engine, a domain-specific model, and a handful of implementation-specific interfaces with the flight software. The only planner input that changes on higher fidelity test-beds is the exact content of the goals provided by on-board systems such as NAV and ACS. This means that modulo timing and performance issues, the planner behavior is expected to behave identically on Unix and the spacecraft for any given input. This permits comprehensive planner testing on Unix platforms with full expectation that the results will scale up to other platforms. A handful of tests are repeated on higher fidelity platforms to address performance issues and to be sure that the plans are identical (i.e., no anomalies are introduced by platform-dependent implementation differences, the CPU, or because we are using a RAMdisk instead of an NFS file system).

5. PLANNER VERIFICATION TOOL

The planner generates a plan from an initial spacecraft state, a set of goals, and constraints. The main requirement on the planner is that the plan meet a long list of correctness requirements. Plans can be several hundred kilobytes long, and are not human-readable. There are about two hundred plans in the test-suite, and the entire suite must be analyzed once a month. There is clearly a need for automated plan verification tools.

We have such a tools for the DS1 planner. It reads the plan into an assertions database and then verify that the assertions satisfy constraints expressed in first order predicate logic (FOPL). This tool was implemented in AP5 [3], a language that supports these kinds of FOPL operations.

The plan correctness requirements are FOPL statements that specify constraints that must hold among plan elements. For example, one constraint is that the plan must not contain OP_NAV_WINDOW tokens if the MICAS camera switch is stuck in the off position (as specified by the MICAS_HEALTH token). This is encoded as the FOPL statement "for all opnav window tokens *w* there exists a MICAS_HEALTH token *h* such that *h* contains *w.*"

Some of these constraints correspond directly to *compatibilities* in the planner model. Other constraints do not map to a single compatibility, but are satisfied by some collection of compatibilities.

The tool is first used to verify that the plans in the test suite satisfy the plan model constraints. This gives us high confidence that the planner model correctly enforces the constraints for all plans. It is much easier to verify that a given plan satisfies a constraint than it is to write compatibilities that enforce the constraint for all plans. The second use is to verify requirements that do not correspond to a single compatibility in the model. These requirements are specified in FOPL.

This tool verifies plans as follows. Roughly speaking, compatibilities are of the form "if token A exists in the plan, then there also exists a token B such that the temporal relation R holds between A and B." A plan satisfies a compatibility if for every token of type A there exists a token of type B in the specified temporal relation, and the relation is specified explicitly in the plan.

The verification tools provide high confidence that the planner generates plans that satisfy the correctness conditions. It is also necessary to validate the correctness conditions themselves. A minimal approach is to have appropriate system engineers review the conditions. We took this approach for RAX. A more systematic approach would be to augment the review process with formal design-validation methods such as SPIN [6] to ensure that the correctness conditions guarantee a small handful of high-level safety and "liveness" conditions.

6. STATUS OF RAX TESTING

As of October 1998, we have performed the low-fidelity test suites on four RAX releases. Between December and March we will perform the system test

[2] Flight processors are typically one or two generations behind the state-of-the-art due to the need for radiation hardening and the need to select a CPU at the beginning of the project (at least two years before launch).

suite on the high-fidelity test beds. We have been able to achieve this testing effort with a four-person testing team working approximately half-time.

7. CONCLUSIONS

Verifying and validating autonomous systems raises a number of issues not faced by traditional flight software. Traditional flight software (FSW) can focus testing efforts on the small handful of known execution paths, whereas autonomous software must provide high confidence that it will behave correctly in all situations. To provide this confidence, the test suite must have adequate coverage of the requirements and input space. Some of the standard coverage metrics and test-suite construction methods are applicable to the RA, but in some cases new metrics and methods were needed. We identified a number of these that we found useful for testing RAX.

Test suites with good coverage also have a relatively large number of test cases, at least with respect to traditional FSW. Since high-fidelity test beds are scarce on flight projects, it was necessary to distribute the tests among high and low fidelity platforms. Several aspects of the Remote Agent architecture made this feasible. We expect that future missions can use a similar approach.

The complexity of autonomous systems makes it is difficult to specify and verify the expected behavior. We identified a number of methods for specifying the expected behavior, and developed tools for automatically verifying the observed behavior against those specifications.

A full-scale testing effort should also include formal validation methods to provide even higher confidence in the RA. We identified a few such methods and performed proof-of-concept demonstrations. Expanding upon these methods is an area for future work.

Overall, the RAX testing effort has identified several issues that arise when testing autonomous spacecraft commanding systems, and demonstrates a credible verification and validation approach that will scale up beyond the scope of the Remote Agent Experiment. Successful validation of the RAX will open the door to use of this exciting technology on future science missions, and perhaps encourage the development of new mission classes that are only possible with autonomous spacecraft.

8. ACKNOWLEDGMENTS

This paper describes work performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration, and by the NASA Ames Research Center. This work would not have been possible without the efforts of the rest of the DS1 team: Doug Bernard, Greg Dorais, Ed Gamble, Bob Kanefsky, Jim Kurien, Nicola Muscettola, Pandu Nayak, Kanna Rajan, Nicolas Rouquette, Will Taylor, and Yu-Wen Tung.

REFERENCES

[1] Doug Bernard, Greg Dorais, Chuck Fry, Edward Gamble Jr., Bob Kanefsky, James Kurien, William Millar, Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Kanna Rajan, Nicolas Rouquette, Ben Smith, and Brian Williams, "Design of the Remote Agent Experiment for Spacecraft Autonomy," In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen CO

[2] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. Nayak, M. D. Wagner, and B. C. Williams, "A Remote Agent Prototype for Spacecraft Autonomy," SPIE Proceedings Volume 2810, Denver, CO, 1996.

[3] D. Cohen. "Compiling Complex Database Transition Triggers,". *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 225-234. Portland, Oregon. ACM Press, 1989.

[4] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation", *IEEE Software*, pp. 83-88, September 1996.

[5] D. M. Cohen, S. R. Dalal, M. L. Friedman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", *IEEE Transactions on Software Engineering*, Vol., 23 No. 7, pp. 437-444, July 1997.

[6] Holzmann, G. Tutorial on SPIN/Promela. *Computer Networks and ISDN Systems*, **25**:981-1017, 1993.

[7] James Clarke "Automated Test Generation from a Behavioral Model", May 1998, Software Quality Week conference, volume 1, section 2T1.

[8] Lowry, M., Havelund, K., and Penix, J., "Verification and Validation of AI systems that Control Deep-Space Spacecraft," in Foundations of Intelligent Systems, Proceedings ISMIS-97: 10th Int'l Symp. Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence, No. 1325, Springer-Verlag, 1997.

[9] Reid Simmons and Greg Whelan "Visualization Tools for Validating Software of Autonomous Spacecraft," In Proc. of the Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS), Tokyo, Japan, 1997